Mastering OpenCL

Session 309

These are confidential sessions—please refrain from streaming, blogging, or taking pictures.

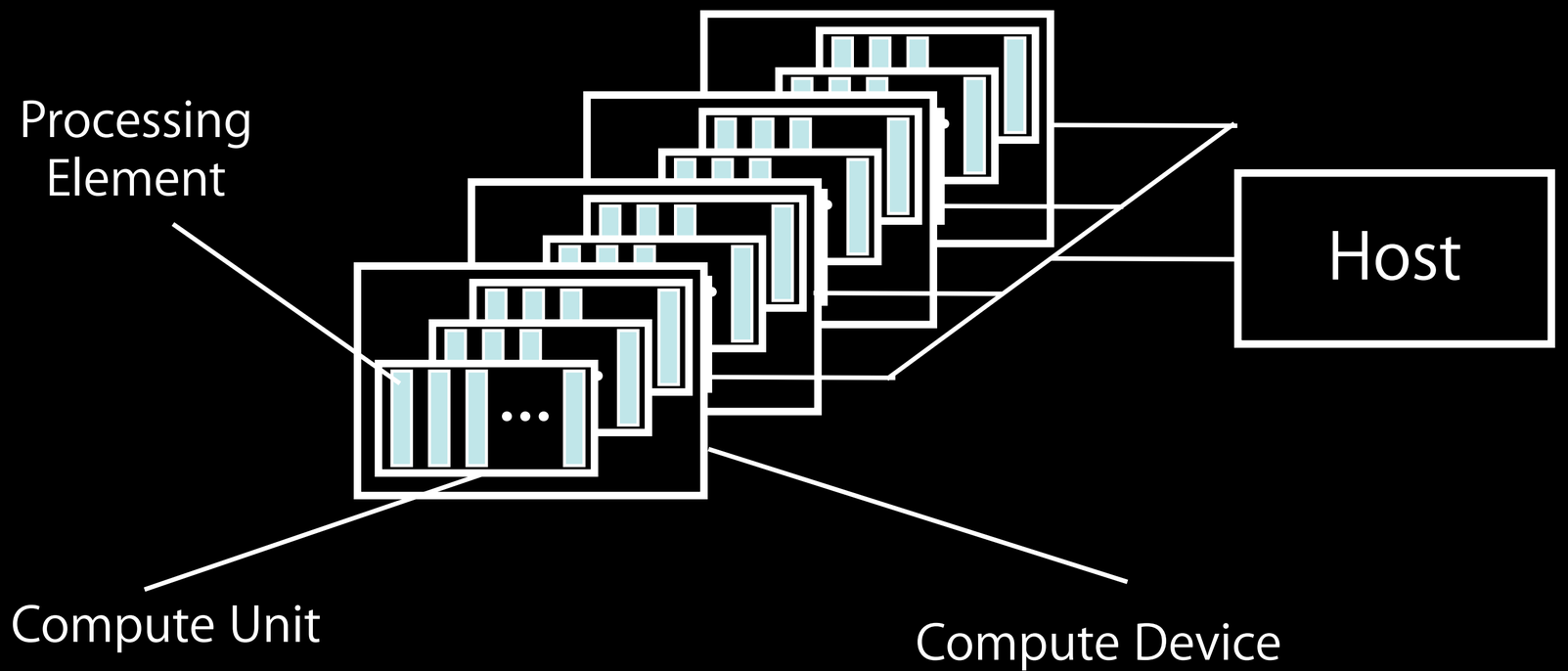
Advanced OpenCL Techniques

Justin Hensley
Office of the CTO
Advanced Micro Devices, Inc.

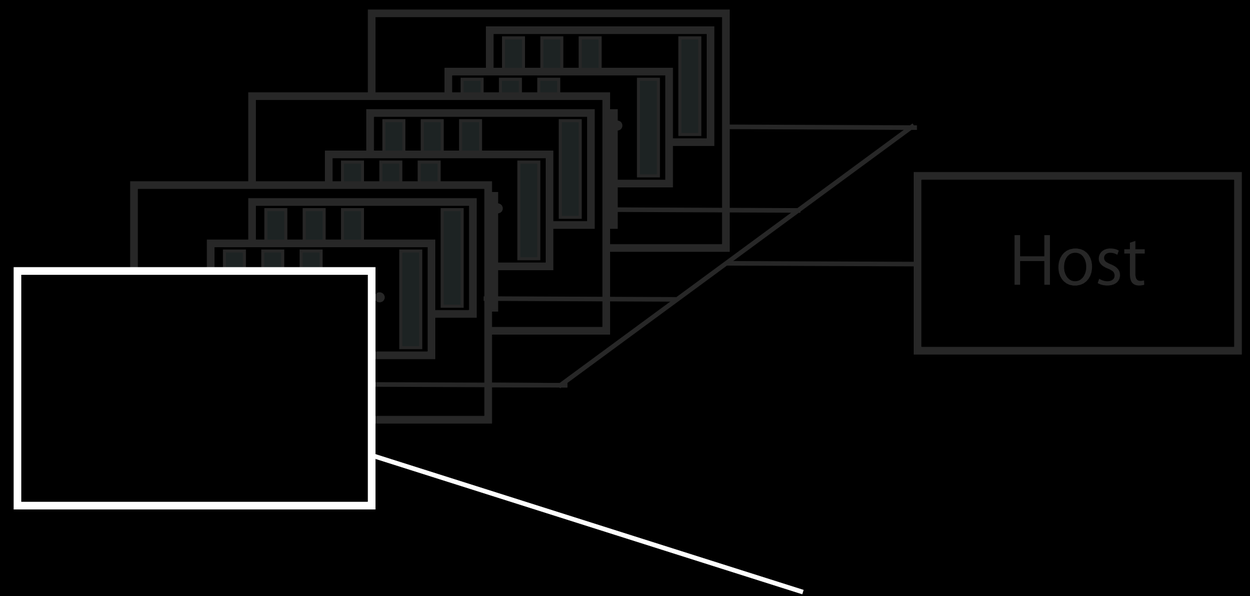
Overview

- Brief overview of ATI Radeon HD 4870 architecture and operation
- “Spreadsheet” performance analysis
- ATI Radeon HD 4870 tips and tricks
- Demo

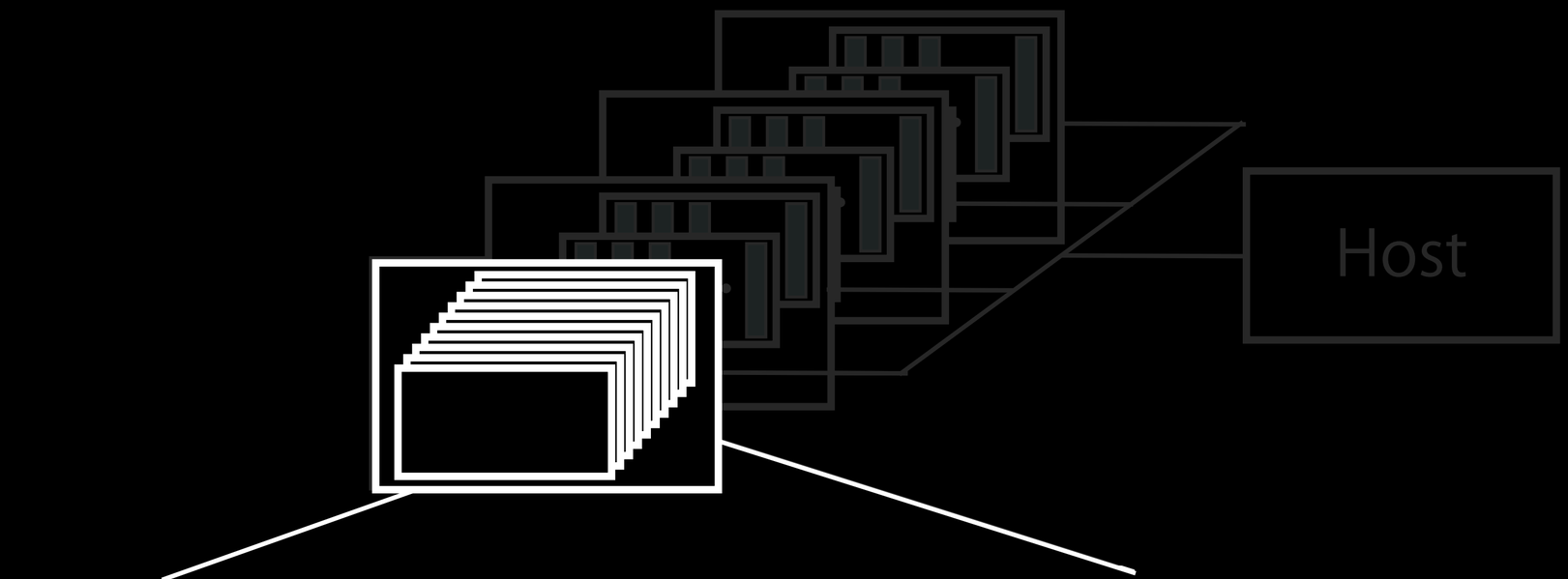
Review: OpenCL View



ATI Radeon HD 4870: Logical View



ATI Radeon HD 4870: Logical View

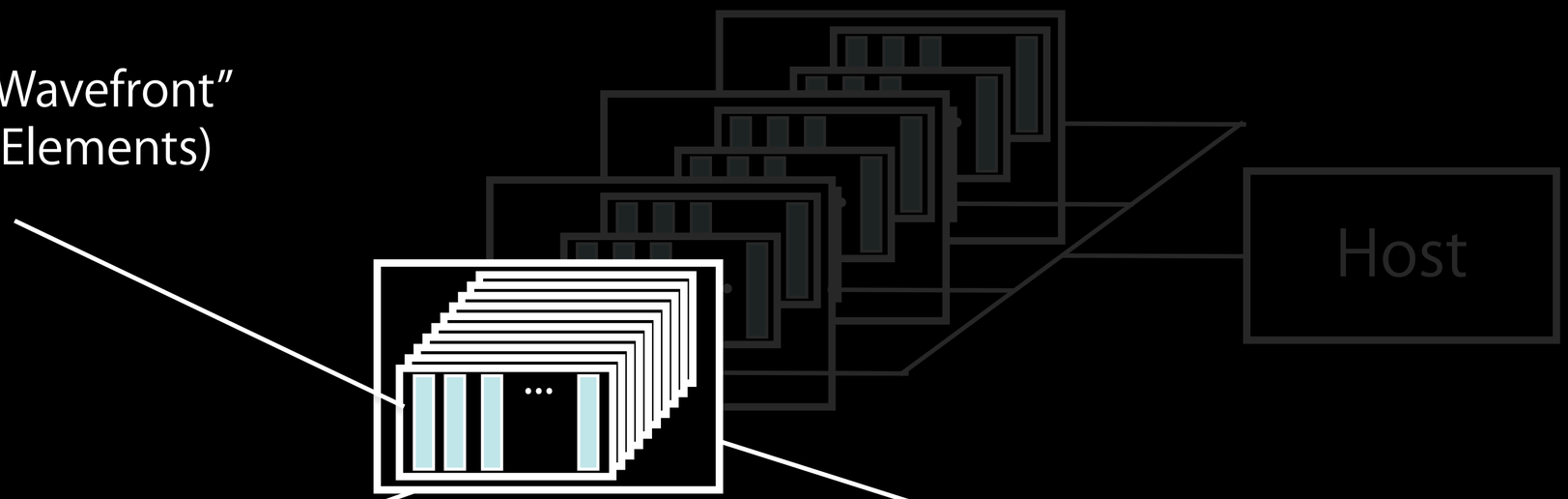


10 SIMDs
(Compute Unit)

Radeon HD 4870
(Compute Device)

ATI Radeon HD 4870: Logical View

64 Element "Wavefront"
(Processing Elements)

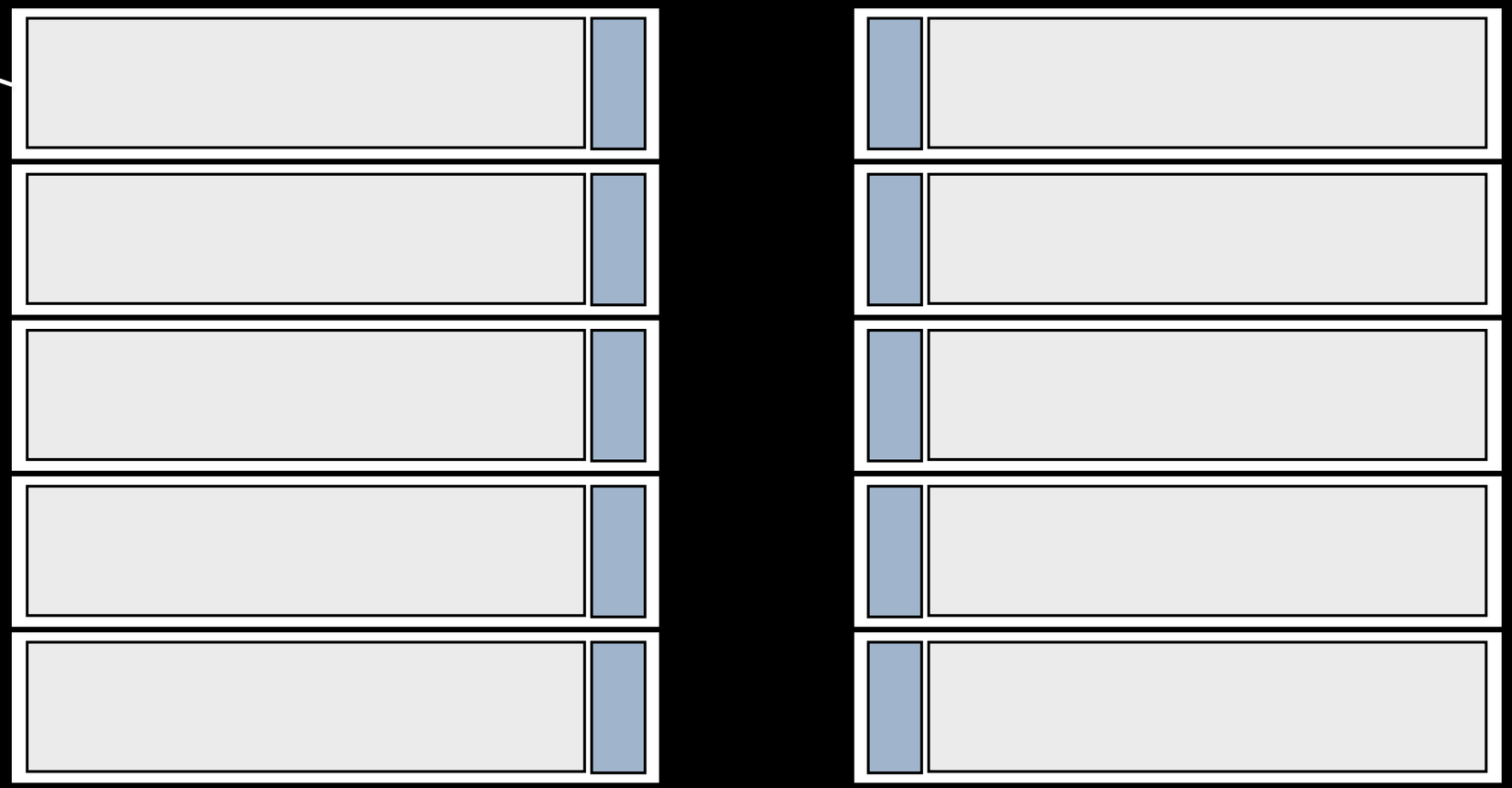


10 SIMDs
(Compute Unit)

Radeon HD 4870
(Compute Device)

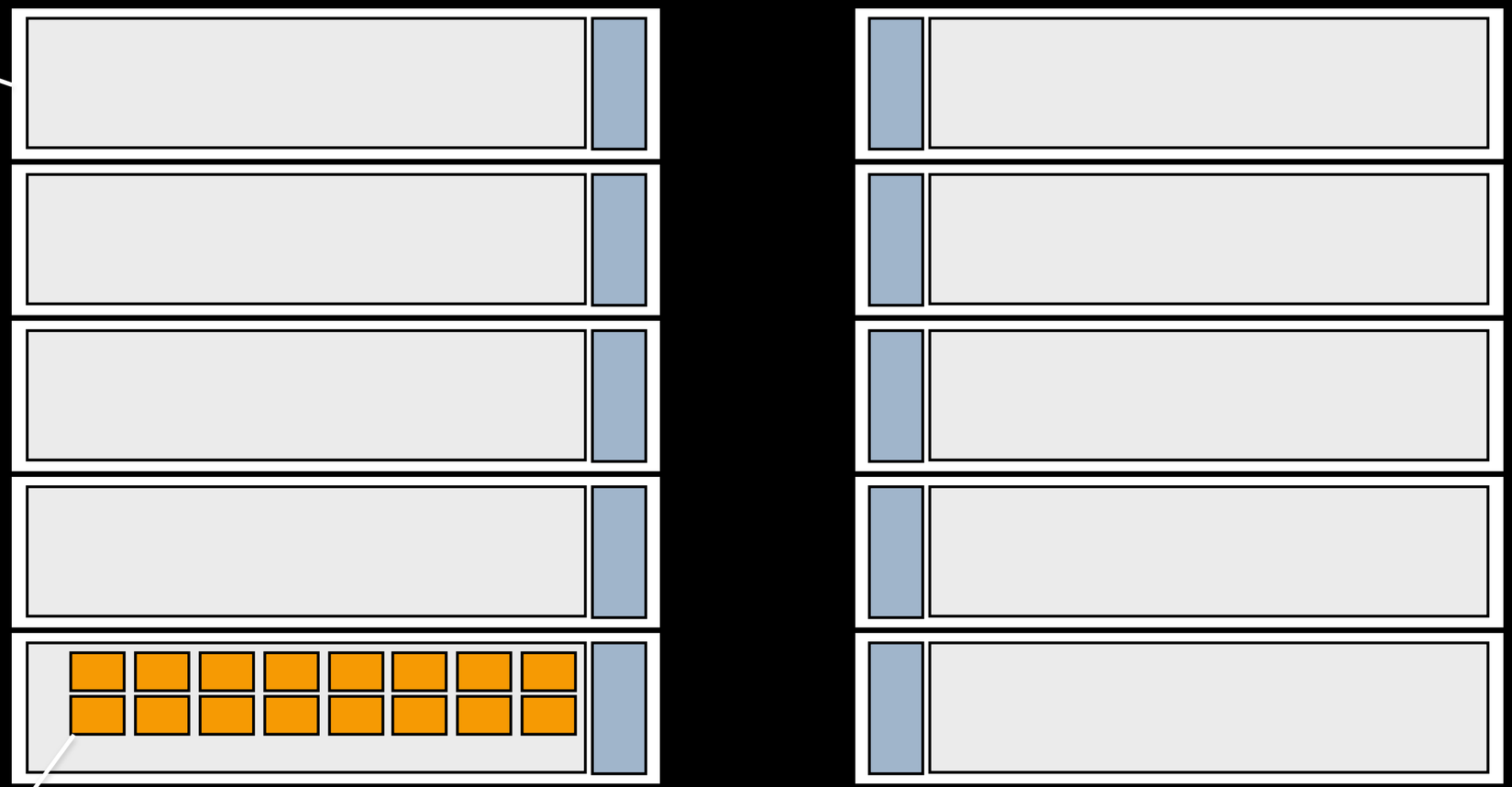
ATI Radeon HD 4870: *Reality*

10 SIMDs



ATI Radeon HD 4870: *Reality*

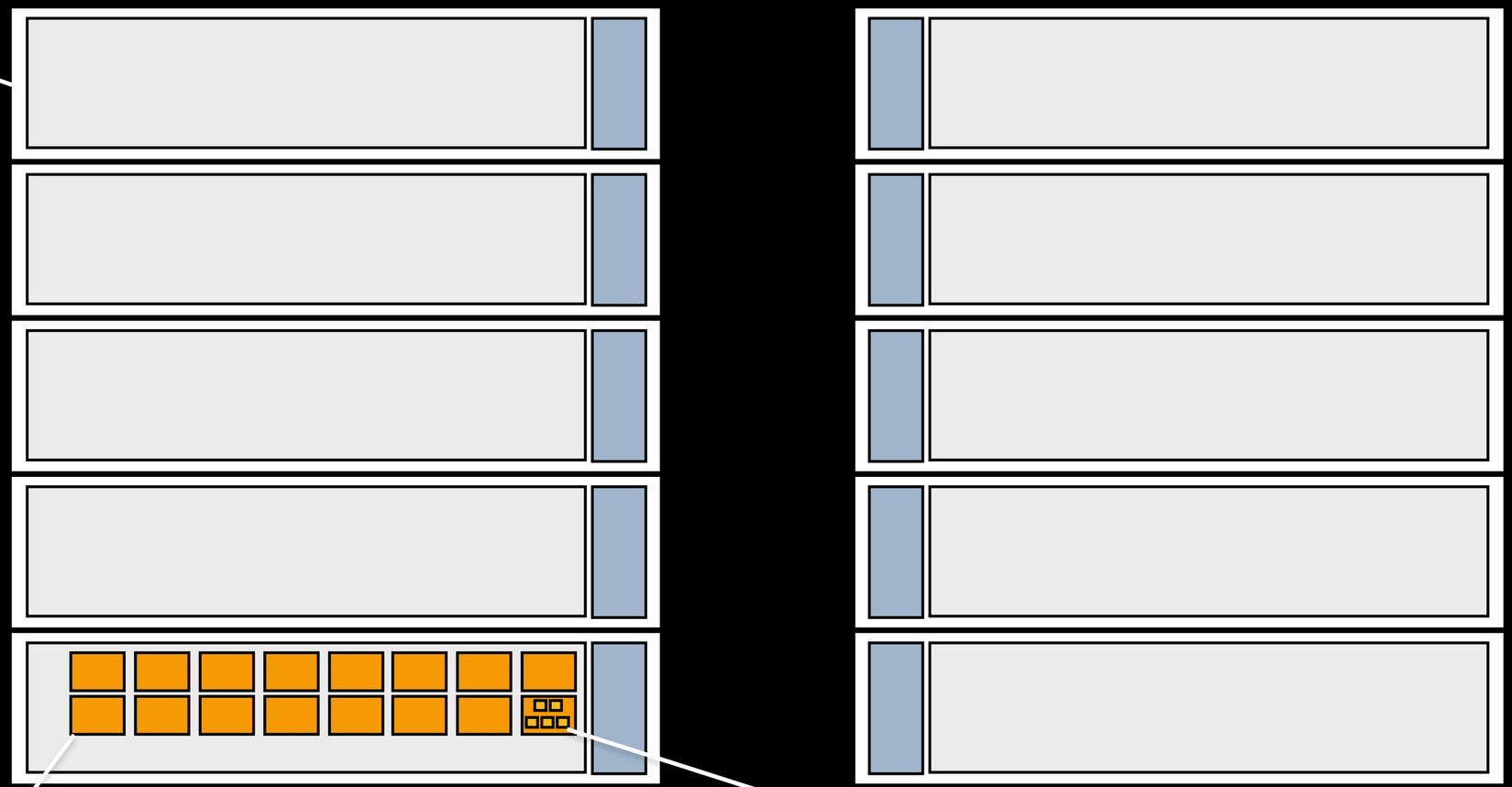
10 SIMDs



16 Processing "Cores" per SIMD
(**64** Elements over 4 Cycles)

ATI Radeon HD 4870: *Reality*

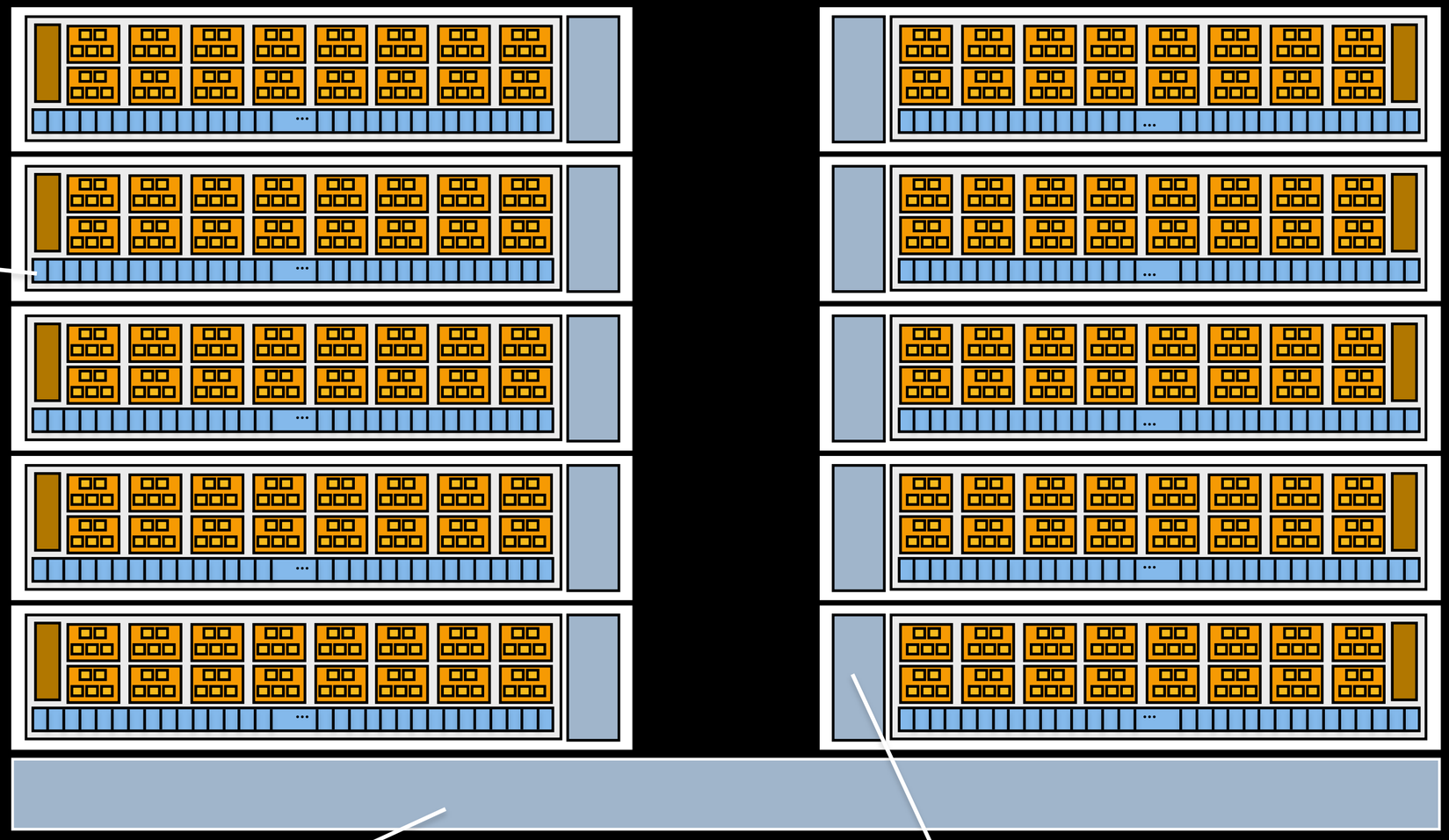
10 SIMDs



16 Processing "Cores" per SIMD
(**64** Elements over 4 Cycles)

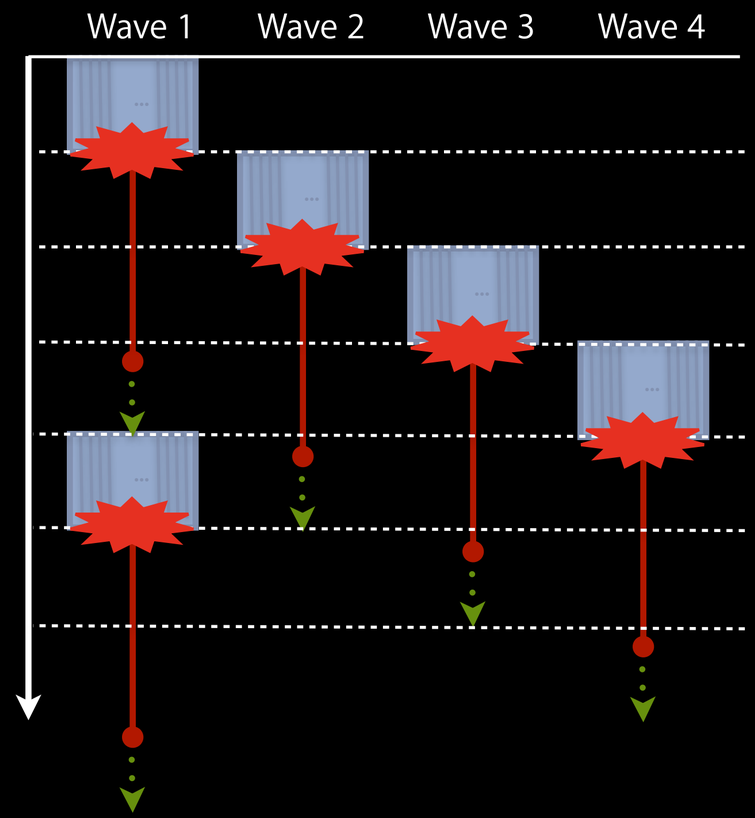
5 ALUs per "Core"
(VLIW Processors)

ATI Radeon HD 4870: *Reality*



Latency Hiding

- GPUs eschew large caches for large register files
- Ideally launch more “work” than available ALUs
- Register file partitioned amongst active wavefronts
- Fast switching on long-latency operations



“Spreadsheet Analysis”

- Valuable to estimate performance of kernels
  - Helps identify when something is “wrong”
- First-order bottlenecks: ALU, TEX, or MEM
  - $T_{kernel} = max(ALU, TEX, MEM)$

$$T_{alu} = \#elements * \#ALU / (10*16) / 750\ Mhz$$

- global_size
- VLIW instructions
- # SIMDs
- “cores” per SIMD
- Engine Clock

Practical Implications on ATI Radeon HD 4870

- Workgroup size should be a multiple of 64
  - Remember: Wavefront is 64 elements
  - Smaller workgroups → SIMDs will be underutilized
- SIMDs operate on pairs of wavefronts

Minimum Global Size on ATI Radeon HD 4870

- **10** SIMDs * **2** waves * **64** elements = **1280** elements
  - Minimum global size to utilize GPU with one kernel
  - Does not allow for any latency hiding!
- For minimum latency hiding: **2560** elements

Register Usage

- Recall GPUs hide latency by switching between large number of wavefronts
- Register usage determines maximum number of wavefronts in flight
- More wavefronts → better latency hiding
- Fewer wavefronts → worse latency hiding
- Long runs of ALU instructions can compensate for low number of wavefronts

Kernel Guidelines On ATI Radeon HD 4870

- Prefer **int4/float4** when possible
  - Processor "cores" are five-wide VLIW machines
  - Memory system prefers 128-bit load/stores
- Consider access patterns—e.g., access along rows
- AMD GPUs have large register files
  - Perform "more work per element"—Example later

Median Filtering

- Nonlinear filter for removing “one-shot” noise in images
- Simple algorithm
    1. Load neighborhood
    2. Sort pixels—Example uses an efficient method
    3. Output median value

Simple Kernel

Compute my index

Handle edge cases

Load neighborhood

Sort along the rows

Sort along the cols

Sort along the diagonal

Output median

```
__kernel void medianfilter_rgba( __global uint *id, __global uint *od, int width, int h, int r )
{
    const int posx = get_global_id(0);
    const int posy = get_global_id(1);
    const int idx = posy*width + posx;

    if( posx == 0 || posy == 0 || posx == width-1 || posy == h-1 )
    {
        od[ idx ] = id[ idx ];
    } else {
       uint row00, row01, row02, row10, row11, row12, row20, row21, row22;
       row00 = id[ idx - 1 - width ];  row01 = id[ idx - width ];  row02 = id[ idx + 1 - width ];
       row10 = id[ idx - 1];           row11 = id[ idx ];          row12 = id[ idx + 1 ];
       row20 = id[ idx - 1 + width ];  row21 = id[ idx + width ];  row22 = id[ idx + 1 + width ];

       // sort the rows
       sort3( &(row00), &(row01), &(row02) );
       sort3( &(row10), &(row11), &(row12) );
       sort3( &(row20), &(row21), &(row22) );
       // sort the cols
       sort3( &(row00), &(row10), &(row20) );
       sort3( &(row01), &(row11), &(row21) );
       sort3( &(row02), &(row12), &(row22) );
       // sort the diagonal
       sort3( &(row00), &(row11), &(row22) );

       // median is the the middle value of the diagonal
       od[ idx ] = row11;
    }
}
```

Auxiliary Functions

Convert to luminance

Swap A and B

Comparison sort

```
float rgbToLum( uint a )
{
    return (0.3f*(a & 0xff) + 0.59*((a>>8) & 0xff) + 0.11*((a>>16) & 0xff));
}

void swap( uint *a, uint *b )
{
    uint tmp = *b;
    *b = *a;
    *a = tmp;
}

void sort3( uint *a, uint *b, uint *c )
{
    if( rgbToLum( *a ) > rgbToLum( *b ) )
        swap( a, b );
    if( rgbToLum( *b ) > rgbToLum( *c ) )
        swap( b, c );
    if( rgbToLum( *a ) > rgbToLum( *b ) )
        swap( a, b );
}
```

More Work per Work-Item

- Prefer read/write 128-bit values
- Compute more than one output per work-item
- Better algorithm (further optimizations possible)
    1. Load neighborhood 8 x 3 via six 128-bit loads
    2. Sort pixels for each of four pixels
    3. Output median values via 128-bit write
- 20% faster than simple method on ATI Radeon HD 4870

Better Kernel

Compute my index

Load row above

Load my row

Load row below

Find four medians

Output medians

```
__kernel void medianfilter_x4( __global uint *id, __global uint *od, int width, int h, int r )
{
    const int posx = get_global_id(0); // global width is 1/4 image width
    const int posy = get_global_id(1); // global height is image height
    const int width_d4 = width >> 2;   // divide width by 4
    const int idx_4 =  posy*(width_d4) + posx;

    uint4 left0, right0, left1, right1, left2, right2, output;
    // ignoring edge cases for simplicity
    left0  = ((__global uint4*)id)[ idx_4 - width_d4 ];
    right0 = ((__global uint4*)id)[ idx_4 - width_d4 + 1];

    left1  = ((__global uint4*)id)[ idx_4 ];
    right1 = ((__global uint4*)id)[ idx_4 + 1];

    left2  = ((__global uint4*)id)[ idx_4 + width_d4 ];
    right2 = ((__global uint4*)id)[ idx_4 + width_d4 + 1];

    // now compute four median values
    output.x = find_median( left0.x, left0.y, left0.z,
                            left1.x, left1.y, left1.z, left2.x, left2.y, left2.z );
    output.y = find_median( left0.y, left0.z, left0.w,
                            left1.y, left1.z, left1.w, left2.y, left2.z, left2.w );
    output.z = find_median( left0.z, left0.w, right0.x,
                            left1.z, left1.w, right1.x, left2.z, left2.w, right2.x );
    output.w = find_median( left0.w, right0.x, right0.y,
                            left1.w, right1.x, right1.y, left2.w, right2.x, right2.y );

    ((__global uint4*)od)[ idx_4 ] = output;
}
```

Demo

- Powdertoy—Mixed fluid/particle simulation
  - Particle state effects fluid state
    - Heat increases pressure
  - Fluid state effects particle state and motion
    - Increased pressure moves fluid

Fluid Kernel Example

```
__kernel void update_air_1( __global float *vx, __global float *vy, __global float *pv)
{
    const int gx = (XRES/CELL);

    const int idx = get_global_id( 0 );
    const int idy = get_global_id( 1 );

    int index = gx * idy + idx;
    float dp;

    if (idx == 0 || idy == 0) { return; }

    dp = 0.0f;
    dp += vx[index-1] - vx[index];
    dp += vy[index-gx] - vy[index];
    pv[index] *= PLOSS;
    pv[index] += dp*TSTEPP;
}
```

OpenCL kernel

```
for(y=1; y<YRES/CELL; y++) {
      for(x=1; x<XRES/CELL; x++) {
          dp = 0.0f;
          dp += vx[y][x-1] - vx[y][x];
          dp += vy[y-1][x] - vy[y][x];
          pv[y][x] *= PLOSS;
          pv[y][x] += dp*TSTEPP;
      }
}
```

Original C Code

Conclusions

- Optimization is a balancing act
- Things to consider
  - Register usage/number of Wavefronts in flight
  - ALU to memory access ratio
    - Sometimes better recompute something
  - Workgroup size a multiple of 64
  - Global size at least 2560 for a single kernel

AMD

Writing OpenCL for Intel CPUs

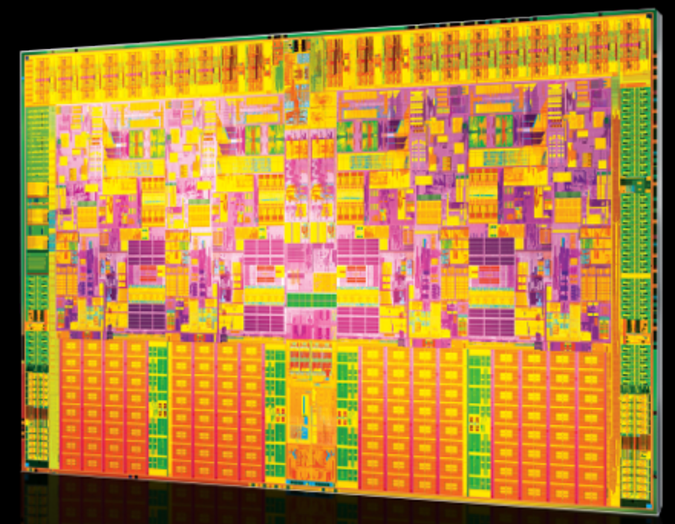
Boaz Ouriel
Senior Software Engineer
Software and Services Group

What Is the Presentation About?

- Writing efficient OpenCL kernels for Intel CPUs
  - Case study
  - Profiling the kernels using Shark
  - DOs and DON'Ts
  - Focus on kernel writing
    - A bit about host code as well
  - Conclusions

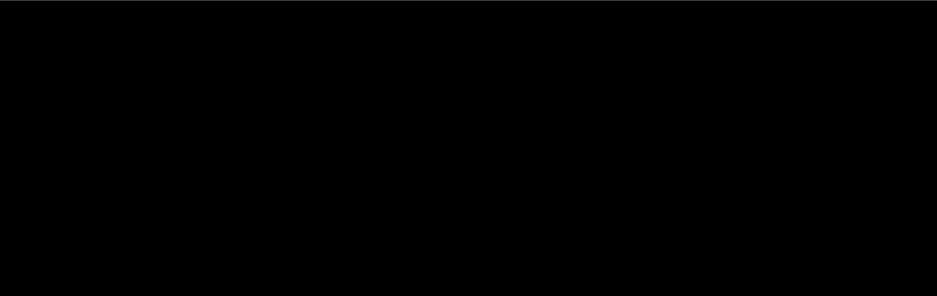
Writing OpenCL for the CPU

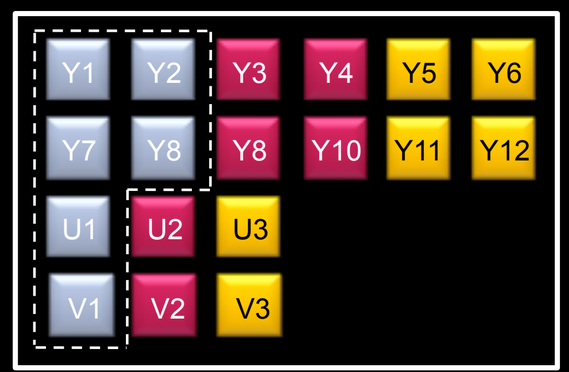
- The challenge of unleashing the performance of modern CPU's
  - Multicore/SMT
  - Vector units (SSE ISA)
- OpenCL is a great framework to harness Intel CPUs
  - Intuitive, easy, and maintainable
  - Unleash the performance of Intel CPUs
    - Multicore
    - Utilize the vector units (SSE ISA)
    - Effective for data parallel and task parallel
  - Portable code across device generations
- Profiling and optimizing CPU kernels is easy
  - Using Shark



The Case Study

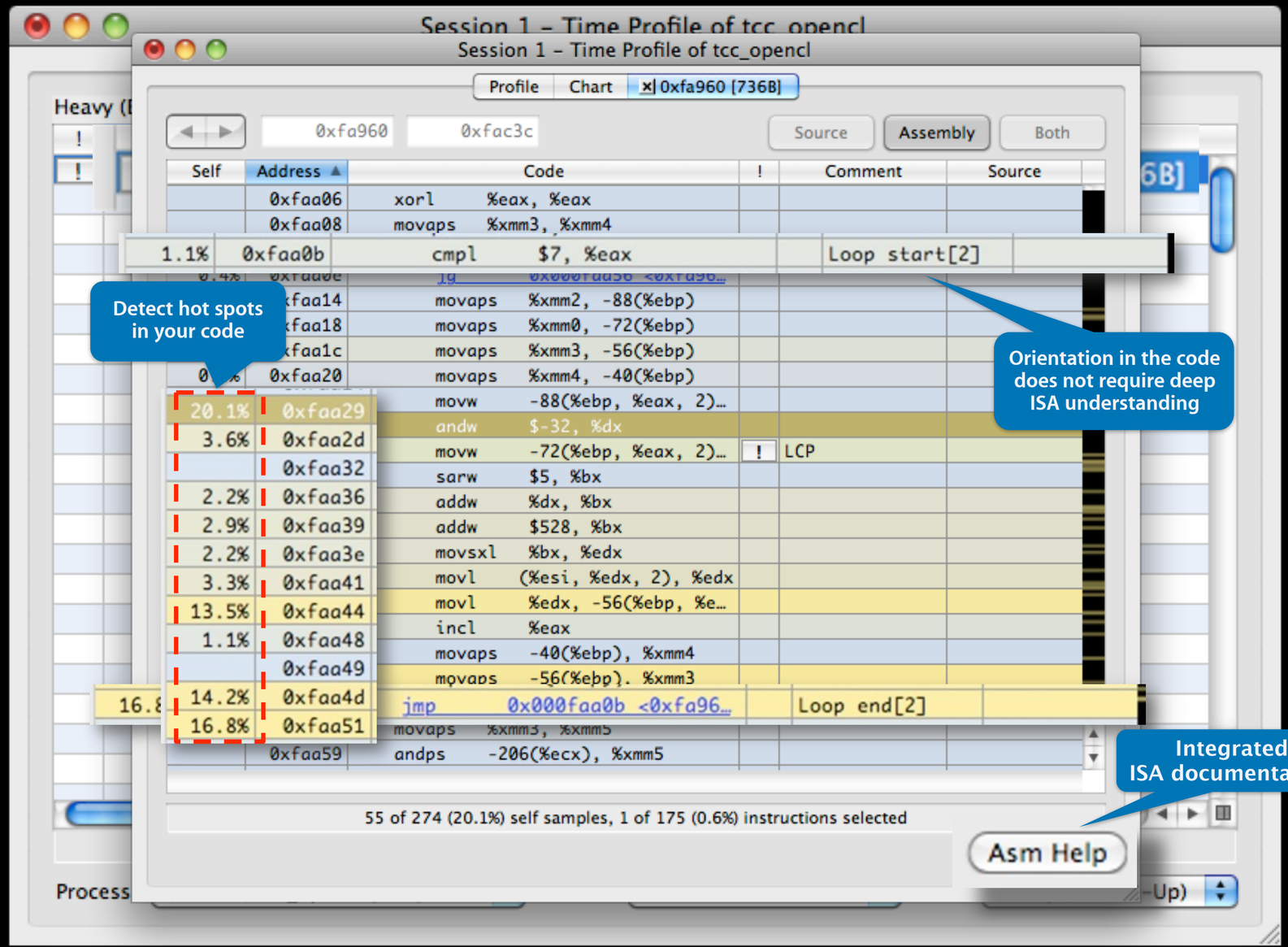
- Two algorithms from the video processing domain
  - Color enhancement
    - Enhance the saturation (color strength) of individual colors
      - Red, Green, Blue, Yellow, Cyan, and Magenta
    - Contrast enhancement
      - Improve extreme dark and bright images
      - Contains serial parts (task parallelism)
- Video frames
  - Processed in YUV 4:2:0 planar color space
  - 10 bits per color component
    - Contained in ***ushort*** (unsigned short)
  - Fixed-point arithmetic
  - Structure of arrays (SOA)



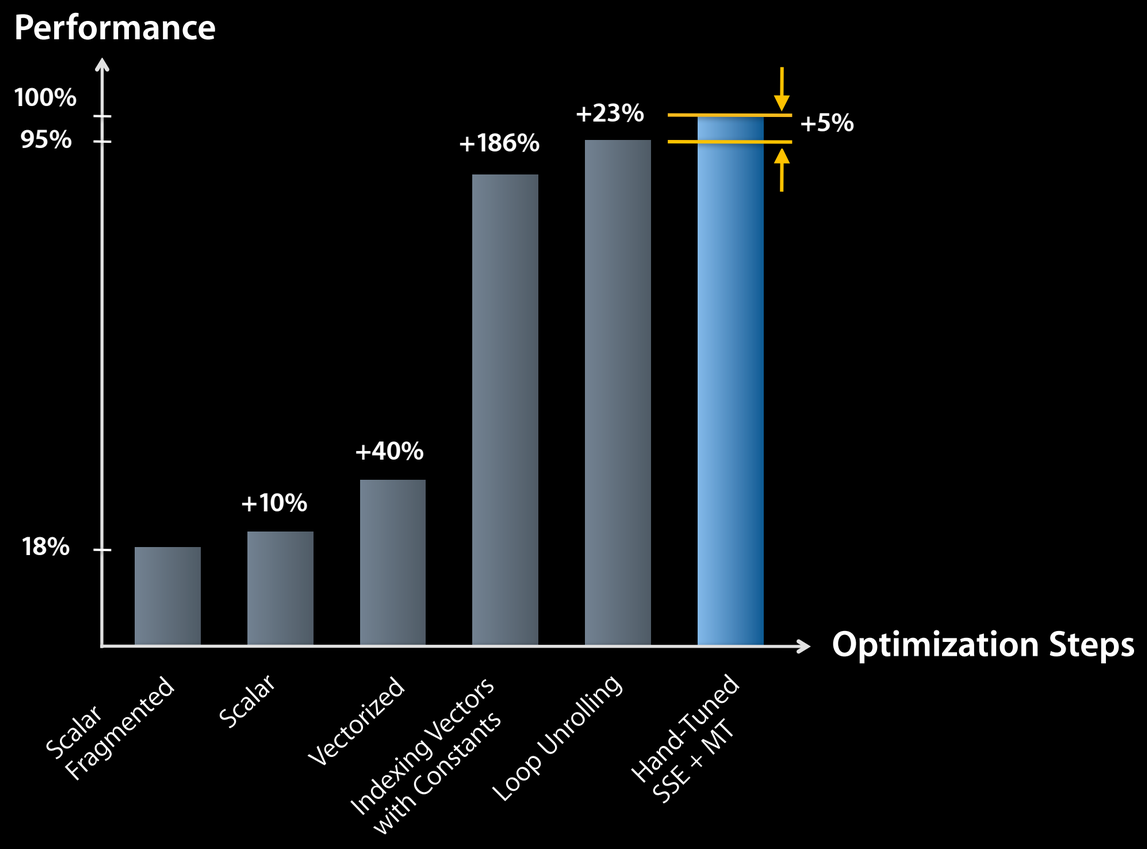


YUV 4:2:0 Frame

Profiling with Shark…



Optimization Steps



Optimization	Description
Scalar Fragmented	• Num work items == Num pixels in frame • Work group size == 1 (limitation)
Scalar	• Num work items == Num Cores
Vectorized	• Using *ushort8* and *ushort16*
Indexing Vectors Using constants	• Replace variable indexing with constant indexing
Loop Unrolling	• Unrolled a small loop

Optimized OpenCL code, on par with hand-tuned code

* Results have been estimated based on internal Intel analysis and are provided for informational purposes only. Any difference in system hardware or software design or configuration may affect actual performance

DOs and DON'Ts When Writing CPU Kernels

DOs

- Match workgroups to cores
  - Set num of WG == num of Cores (use query)
  - Remember: WG size is one (current restriction)
- Vectorize your code
- Index vectors using constants
  - Where not possible, cast vectors to arays
- Loop unrolling can help
  - Small loops
- Additional
  - Use of Prefetch built-in function can help
  - "*do-while*" is better than "*for*" loops

DON'Ts

- Partition your work to fragments
  - Don't oversubscribe
  - Usually not optimal for balanced kernels
- Dynamic vector indexing is expensive
  - Try avoiding accesses using variables
- Vec. { hi | lo | odd | even } are expensive

Conclusions

- OpenCL is a great framework to harness Intel CPUs
  - Intuitive, easy, and maintainable
- OpenCL can help you generate optimized code for the CPU
  - Scales well across cores
  - Effectively utilizes SSE ISA
  - Suits well serial parts using task parallelism
  - Manually done today
- Portable code
  - Across devices
  - Across device generations

Demo

Optimizing OpenCL for NVIDIA GPUs

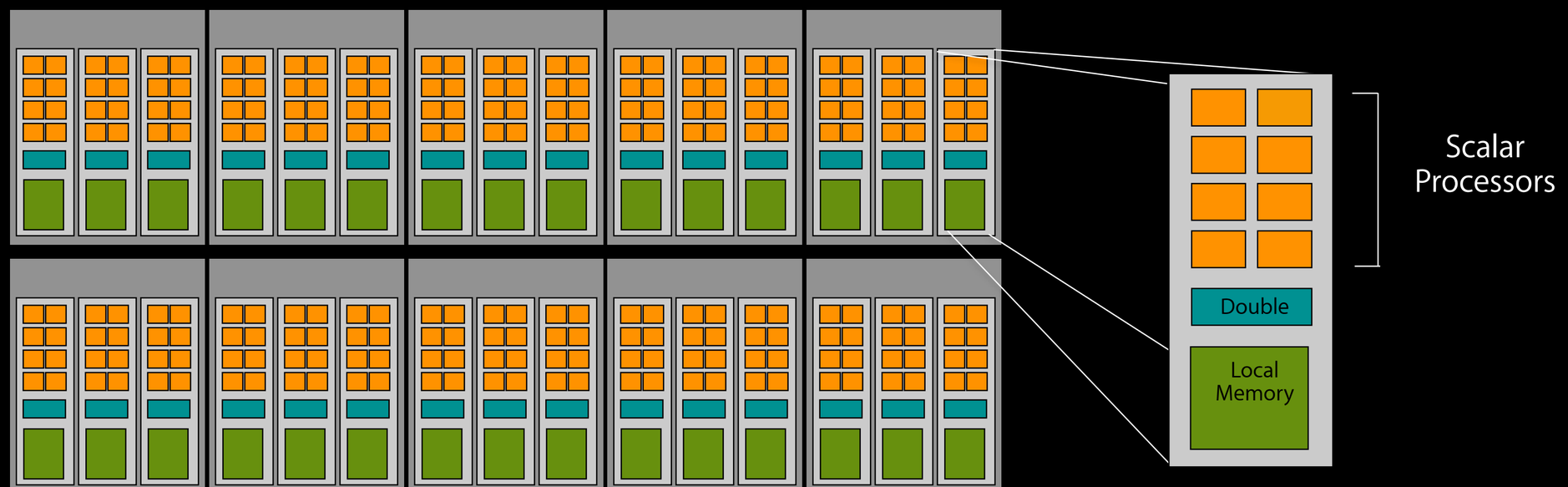
Chris Lamb
NVIDIA

Outline

- Hardware
- Memory optimizations
- Execution configuration optimizations
- Instruction optimizations
- Demo

NVIDIA 10-Series Architecture

- 240 scalar processor (SP) cores execute threads (work-items)
- 30 streaming multiprocessors (SMs) exposed as OpenCL Compute Units, each of which contain
  - 8 scalar processors
  - 2 special function units (SFUs)
  - 1 double precision unit
  - __local memory that enables work-item cooperation



Optimize Algorithms for NVIDIA Hardware

- Maximize independent parallelism
- Maximize arithmetic intensity (math/bandwidth)
- Sometimes it's better to recompute than to cache
  - GPU spends its transistors on ALUs, not memory
- Do more computation on the GPU to avoid costly data transfers
  - Even low parallelism computations can sometimes be faster than transferring back and forth to host

Outline

- Hardware
- Memory optimizations
- Execution configuration optimizations
- Instruction optimizations
- Demo

Optimize Memory Access

- Coalesced vs. noncoalesced = order of magnitude
  - Access pattern of sequentially numbered work-items
  - First order performance effect
- Optimize for spatial locality of accesses in cached texture memory (OpenCL Images)
  - Image reads may benefit from processing as 2D blocks
  - Experiment with work-group aspect ratio to discover what's best
- Let OpenCL allocate memory (CL_MEM_ALLOC_HOST_PTR)
  - The implementation can optimize alignment and location
  - Can still get access for the host via clEnqueueMap{Buffer|Image}

Take Advantage of __local Memory

- Hundreds of times faster than __global memory
- Work-items can cooperate via __local memory
  - Barrier() only needs CLK_LOCAL_MEM_FENCE, which is much lower overhead
- Use it to avoid noncoalesced access
  - Stage loads and stores in local memory to reorder noncoalesceable addressing
- For __local memory, avoid high-degree bank conflicts
  - Generally a far second order effect after other memory optimizations

Outline

- Hardware
- Memory optimizations
- Execution configuration optimizations
- Instruction optimizations
- Demo

Use Parallelism Efficiently

- Partition your computation to keep the GPU multiprocessors equally busy
  - Many work-items, many workgroups
  - Work-groups >> number of compute units
- Keep resource usage low enough to support multiple active workgroups per multiprocessor
  - Registers, __local memory can reduce available parallelism

Outline

- Hardware
- Memory optimizations
- Execution configuration optimizations
- Instruction optimizations
- Demo

Scalar Architecture

- NVIDIA GPUs have a scalar architecture
  - Use vector types in OpenCL for convenience, not performance
  - Generally want more work-items rather than large vectors per work-item
- Maximize use of high-bandwidth memory
  - Maximize use of __local memory
  - Minimize accesses to __global memory
  - Maximize coalescing of __global memory accesses
- Optimize performance by overlapping memory accesses with hardware computation
  - High arithmetic intensity programs (i.e., high ratio of math to memory transactions)
  - Many concurrent work-items

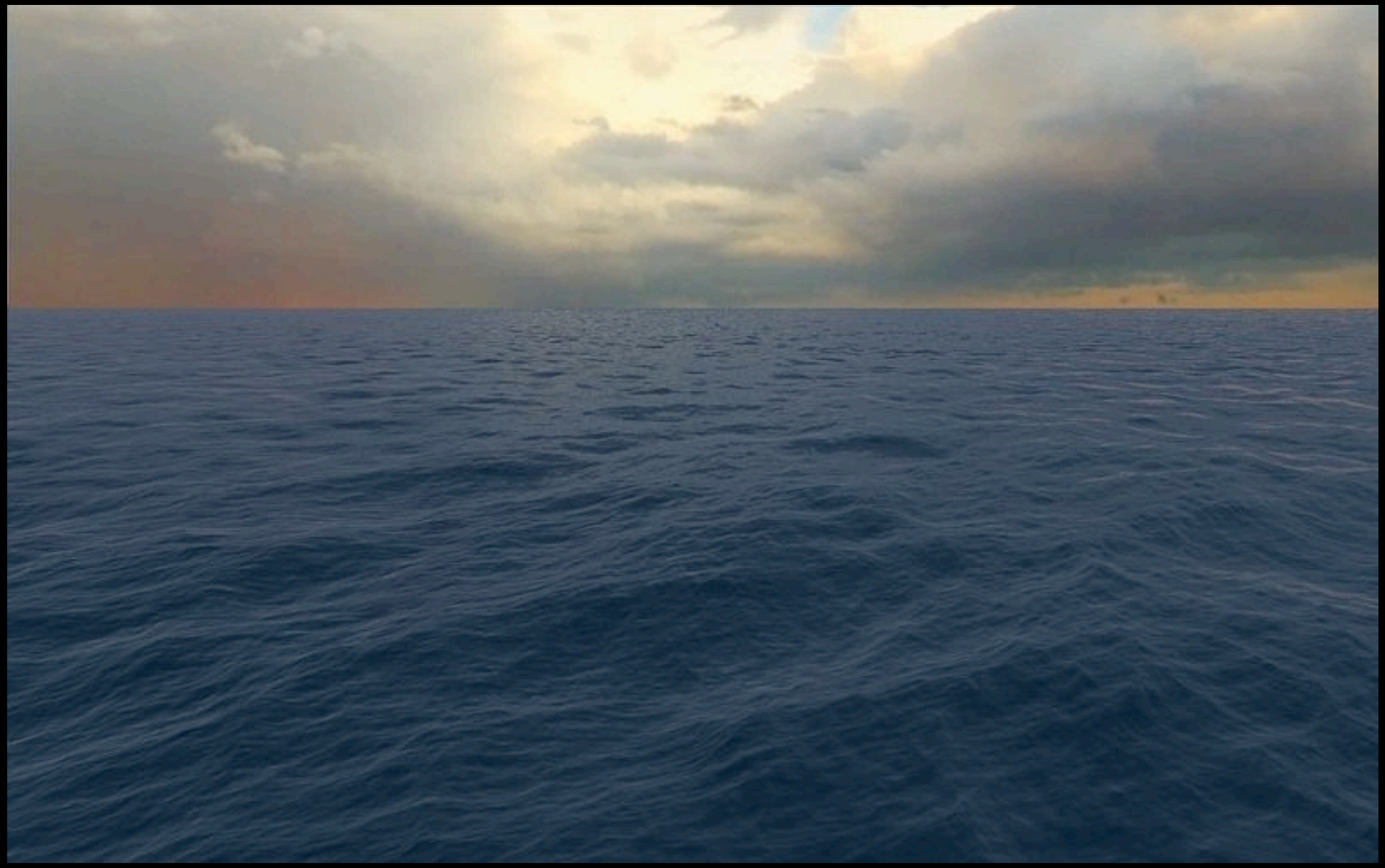
Math Library and Compiler

- Use native_* and half_* math functions where possible
  - Many have a direct mapping to hardware ISA
  - Can be orders of magnitude faster than higher precision variants
  - Note that half_* functions do not mean the FP16 type extension
- Use the -cl-mad-enable compiler option
  - Permits use of FMADs, which can lead to large performance gains
- Investigate using the -cl-fast-relaxed-math compiler option
  - Enables many aggressive compiler optimizations

Outline

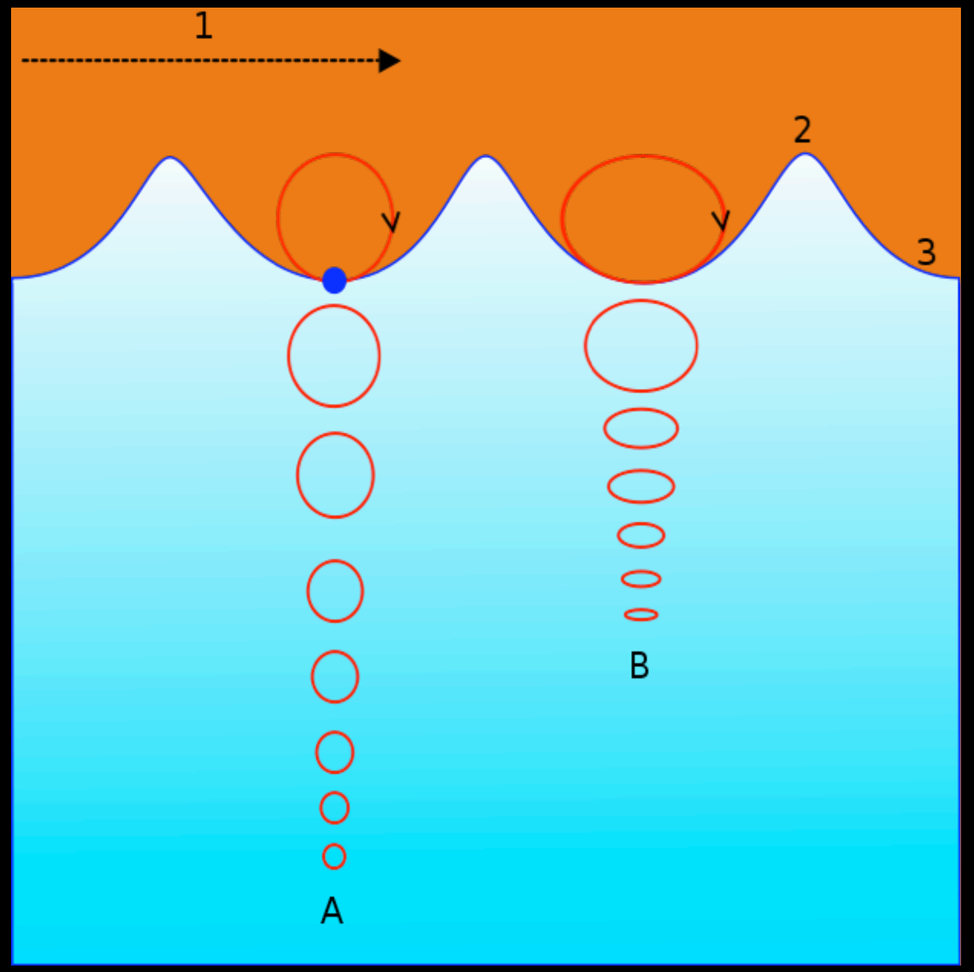
- Hardware
- Memory optimizations
- Execution configuration optimizations
- Instruction optimizations
- Demo

Ocean Surface Simulation



The Algorithm: Wave Composition

- The ocean surface is composed by enormous simple waves
- Each sine wave is a hybrid sine wave (Gerstner wave)
  - A mass point on the surface is doing vertical circular motion



Simulation Overview

- Based on Jerry Tenssendorf's paper "Simulating Ocean Water"
  - Statistic-based, not physics-based
  - Generate wave distribution in frequency domain, then perform inverse FFT
  - Widely used in movie CGIs since 1990s, and in games since 2000s
- In movie CGI: The size of height map is large
  - *Titanic*, 2048 x 2048
  - *Waterworld*, 2048 x 2048
- In games: The size of height map is small
  - *Crysis*, 64 x 64
  - *Resistance 2*, 32x32
  - All simulated on CPU (or Cell SPE)

Performance Issues

- The simulation is required to generate the displacement map in real time
- Computing FFT on CPU becomes the bottleneck when the displacement map gets larger
  - Larger texture also takes longer time on CPU-GPU data transfer
  - However, large displacement map is a must-have for detailed wave crests
- GPU computing is really good at FFT
  - Multiple 512 x 512 transforms can be performed in trivial time on high-end GPUs
  - Multiple 1024 x 1024 transforms are affordable for high-quality, real-time rendering

More Information

Allan Schaffer
2D and 3D Graphics Technologies Evangelist
aschaffer@apple.com

Labs

OpenCL Lab	Graphics and Media Lab B Thursday 3:30PM

Q&A

WWDC Session Survey

You can rate your sessions on the WWDC Attendee Site.

Survey forms are located on each session description page at:

developer.apple.com/wwdc/attendee

WWDC09